BAKKALAUREATSARBEIT

Graphical Development Environment for Embedded Systems

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Bakkalaureus der Technischen Informatik

unter der Leitung von

Univ.Ass. Dipl.-Ing. Dr.techn. Wilfried Elmenreich
Institut für Technische Informatik 182

durchgeführt von

Philipp Jahn und Thomas Polzer
Matr.-Nr. 0325871, 0325077
Wien, Baden

Wien, im September 2006 .....

Graphical Development Environment for Embedded Systems

A new tool, *Graphical Microcontroller Programing (GMP)*, for programming embedded systems graphically is presented in this thesis. The main goal at the moment is to focus on 8-bit AVR controllers. Current graphical development tools in this area have certain advantages and disadvantages. We want to stress them out and show our design approach. As a result of this thesis, we have developed a tool (*GMP*) to provide a dynamically model-based approach, which should allow advanced users to build their own libraries for certain controllers, and help novice users in getting started with the programming of microcontrollers.

Contents

1 Introduction

The main issue of this thesis is to program embedded systems by means of a graphical development environment. We will focus especially on simple low cost 8-bit microcontrollers like the Atmel AVR series. We want to analyze existing tools like the Generic Modeling Environment (GME) in combination with GRATIS II, which provides a generic graphical environment, a parser and a code generator for TinyOS. We also want to take a look on special AVR tools as well as extensive and powerful products like Simulink and LabVIEW that might be able to be adopted for our task.

The reason for this research is to avoid the intrinsic complexity of programming embedded systems for implementing standard tasks on microcontroller systems. This should be possible per drag and drop without writing a single line of code. The big challenge is, that we need a fully dynamic environment to be flexible enough to implement more complex components and not being restricted to a limited number of possible applications or being limited to one type of microcontroller.

Since the major part of microchips are used within embedded systems and not for general purpose computers[1], we assume that a tool which simplifies the process of developing embedded systems will have a major impact on productivity.

1.1 Motivation and Objectives

Nowadays it becomes more and more important to speed up the development processes in all branches of business to be competitive in commerce. Especially embedded system development processes are more important and more complex than ever before. A graphical development environment would be able to abstract the development from the embedded target and make it possible for the developer to focus on the developing process rather than the internals of the target. Historically there have not been any graphical tools in this area. At the very beginning machine code was written. The next step was to use an assembler language to specify the applications. Today there are a lot of good

[1] http://www.embedded.com/1999/9905/9905turley.htm

C compilers for microcontrollers available, so you can write C code without generating too much overhead. In the last few years some basic approaches for easier development with different kinds of graphical programming aspects came up, which we will present in the next chapter [vH04b, vH04a].

The crucial points in the field of embedded systems are limited memory, small or non existent long term storage, changing the target systems, debugging and testing. If you want to use a certain operating system (OS) you first have to check if the target system has enough memory, adept the OS and optimize the code for size. If you change the target system you might have to reconfigure all input and output ports and change the setup of the peripherals. The most difficult part in embedded system development is debugging and testing within the limited capabilities of the controllers. Normally you have to work with outputs over a serial line, LEDs or use an oscilloscope or logic analyzer to figure out where to find the problems of your program. Often only one wrongly set bit can lead to an error that is very hard to track. If you are using an OS you can take advantage of stable and tested libraries but the development is still very complex. Very often you depend on vendor specific tools, which makes the code unportable. There are also some controllers which do not have enough memory or have some other limitations, making it impossible to use any OS [vH04a].

All these complex tasks can be easily solved by experts in this area. Our objective is to develop a software which makes it easier to develop, debug and test embedded software. Today's personal computers are not feasible to be used in an office without a graphical user interface (GUI). Although the real power of the software is the OS itself and not the GUI, it is the graphical part which makes it possible for users to handle certain tasks easier, faster and with much less initial training. Since the same is true for developing embedded system software with graphical tools we can use this platform to eliminate much of the complexity of the development process [vH04c, vH04a].

This complexity originates in the numerous steps you have to go through developing an embedded application. Normally you use an editor to write the source code into multiple files. These files must be compiled into object files and linked together with additional libraries. The result is an executable which have to be downloaded to the target. It is followed by debugging and testing. Additionally these steps are repeated in multiple iterations. There are some programs like *make* under Unix, which helps to automate and unify the compilation process through a centrally controlled *Makefile*. But still you have to get used to the handling of Makefiles, which becomes more and more difficult with increasing complexity of the applications. Nevertheless, the make tool is seen as an early milestone in simplifying the creation of embedded applications.

We want to help developers to focus on their projects and the features of the

embedded application. We believe that a graphical environment is the easiest way for it. Such an environment should be easy to understand, intuitively handled and easy to configure. Furthermore it is often necessary to optimize the code not only for speed, but primarily for size. Another requirement is the possibility to change the desired target architecture easily.

1.2 Structure of the Thesis

The thesis is structured as follows: Chapter 2 gives an introduction to the concepts of our research and shows some different basic approaches that are currently available. An overview of those different approaches is listed in Chapter 3 and they are compared to our requirements. In Chapter 4 and 5 we explain our chosen design approach and our implementation. Chapter 6 describes the results and success of our research. The thesis finally ends with Chapter 7 as a conclusion and outlook to future research in the field of graphical development environment for embedded systems.

2 Concepts and Challenges

We want to provide a graphical development environment for embedded systems. It should be possible to easily use the components provided by a microcontroller (for example: Timer, I/O-Ports, ADC, Interrupts, ...). This components should be available in a palette and dragged and dropped into the editor. The contents of this palette should be dynamically. This means, that it will be possible to add new components without changing the application code. Therefore it is necessary to have a dynamic model. Components are connected together using connections between certain input and output ports. Two kind of connections are available (data and event connections). They represent the data and control flow in the embedded application.

Out of this graphical design we have to generate code which will be downloaded to the target controller. We have to choose between two basic approaches. The first one is to generate code for a certain embedded operating system, like Embedded Linux. Therefore, we have to download the code and the binaries of the operating system to the target. The other approach is to generate simple C code. This code is compiled into machine code and downloaded to the target. Thus, existing tool chains based on C code can be used to compile and download the program to the target system.

Other basic questions are:

- Should the components be developed statically or dynamically? The later would provide a modular approach and facilitate professionals to widen the components palette by defining super blocks or developing specific components they need for certain tasks or microcontrollers.
- Using an event driven approach seems much more intuitive for the users than a time driven one.
- Another point is the implementation of state machines and the usage of object oriented code. This questions will be handled in future work only.
- Is it possible to use this concept also for distributed embedded systems? An important question is, if it is possible to implement a TTP/A based distributed application using the graphical development environment. This question will also be handled in the future.

3 Related Work

Because of the many aspects that work together in this thesis we give a wide overview of different environments which helps speed up the development process in the field of embedded systems.

3.1 GRAPE

There are not many references for this tool, but we found a news entry on the internet which linked to a company named grapesys in Israel. We contact Ori Idan, one of the developers, and he said that GRAPE is unfortunately not under development any more. Because it seemed to be very close to our objectives, we describe the main principle of this tool. GRAPE was an Graphical-programming tool for 8-bit microcontrollers. Those were the features supported by GRAPE [Ltd02]:

- Shortened software development time, reducing time to market of new products
- full graphical development system, including a graphical editor, compiler and code generator
- Graphical simulator for debugging
- Expandable library of predefined modules
- Code generator for: Motorola HC08, PIC-16Cxxx, PIC-16Fxxx and 8051 derivatives

The tool is described as easy-to-use graphical programming tool, where you only have to draw a block diagram of the application. The software compiles it directly to the target microcontroller's machine code and optimize it [Ltd02].

Co-Founder and CEO of D. S. Grape, Ori Idan, commented in 2002: *"This new tool integrates the entire software production process under one convenient framework. We expect that GRAPE will enable hardware engineers and software developers to easily and efficiently generate microcontroller applications [Ltd02]."*

During our conversation with Mr. Ori Idan he told us some more important details of their approaches: *"We did not use any OS in 8 bit microcontrollers as there is not enough memory for an OS image. We did not use C either, we directly created the machine code of the target controller, this allowed us for the optimization we needed for memory size."*

3.2 Eclipse, EMF, and GEF

Eclipse itself is not a Graphical Development Environment (GDE) and not directly involved in the graphical development process, but there exists several plug-ins we can use for modelling and editing graphically. For example, EMF and GEF which are explained in the next sections.

Eclipse is an open source community whose projects are focused on providing a vendor-neutral open development platform and application frameworks for building software. The Eclipse Foundation is a not-for-profit corporation formed to advance the creation, evolution, promotion, and support of the Eclipse Platform and to cultivate both an open source community and an ecosystem of complementary products, capabilities, and services. As it says in the Purposes section of the Foundation's Bylaws: The purpose of Eclipse Foundation Inc.,(the "Eclipse Foundation"), is to advance the creation, evolution, promotion, and support of the Eclipse Platform and to cultivate both an open source community and an ecosystem of complementary products, capabilities, and services. Eclipse has formed an independent open eco-system around royalty-free technology and a universal platform for tools integration. Eclipse based tools give developers freedom of choice in a multi language, multi platform, multi vendor environment. Eclipse provides a plug-in based framework that makes it easier to create, integrate and utilize software tools, saving time and money. By collaborating and exploiting core integration technology, tool producers can leverage platform reuse and concentrate on core competencies to create new development technology. The Eclipse Platform is written in the Java language and comes with extensive plug-in construction toolkits and examples. It has already been deployed on a range of development workstations including Linux, HP-UX, AIX, Solaris, QNX, Mac OS X and Windows based systems [www.eclipse.org].

3.2.1 Eclipse modelling Framework (EMF)

EMF is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce

a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. Models can be specified using annotated Java, XML documents, or modeling tools like Rational Rose, then imported into EMF. Most important of all, EMF provides the foundation for interoperability with other EMF-based tools and applications [www.eclipse.org/emf].

3.2.2 Graphical Editing Framework (GEF)

GEF allows developers to create a rich graphical editor from an existing application model. GEF consists of 2 plug-ins. The org.eclipse.draw2d plug-in provides a layout and rendering toolkit for displaying graphics. The developer can then take advantage of the many common operations provided in GEF and/or extend them for the specific domain. GEF employs an MVC (model-view-controller) architecture which enables simple changes to be applied to the model from the view [www.eclipse.org/gef].

GEF is completely application neutral and provides the groundwork to build almost any application, including but not limited to: activity diagrams, GUI builders, class diagram editors, state machines, and even WYSIWYG text editors [www.eclipse.org/gef].

3.3 TimeSys TimeStorm Integrated Development Environment (IDE)

The TimeStorm IDE is a collection of plug-ins for Eclipse that can be easily integrated into the framework. The reason to decide for Eclipse is the openness, flexibility and extensibility, so it is possible that other companies can take advantages of already developed plug-ins and concentrate on their special task. TimeStorm IDE is not an example of a GDE but it has to be mentioned because it is the connective link between developing with different types of programs (editor, debugger, download-tool, test routines, shells, make, etc.) and a GDE. They see the IDE as the next leap after the make tool, which helps to automate the development process. They focusing on one consistent, graphical interface in all phases of this process. So they provide plug-ins for compilation, deployment, debugging, testing within the same graphical interface. The plug-ins accordingly to Eclipse seems to be one single application, that serves as control center for the whole development process to increase the productivity of software developing, especially in the field of embedded systems [vH04a]. Some of the basic features of the TimeStorm IDE are [vH04a]:

- A central directory, called workspace, which contains all files and configurations for that project
- Selecting and changing the used toolchain at any stage of the project, that makes it easy to develop on multiple embedded systems.
- Automatically generating and maintaining the Makefiles, which could be a very complex task for larger projects.
- Integrate custom commands and so override default behavior of automation processes provided by TimeStorm IDE.
- One application that can be used for all stages of the development cycle.
- Using different Run/Launch configurations for different targets, debugging, monitoring and testing.

3.4 TimeSys TimeStorm Linux Development Suite

The TimeStorm Linux Development Suite provides the integration of cross-compilers, simple execution, debugging configurations and also automate transferring applications to the target board. TimeStorm uses embedded Linux as operating system on the targets. Linux is the operating system of choice of most new embedded development projects. Because of the many Linux distributions that are available today, TimeStorm did not choose a particular distribution but can handle distributions from any vendor. It is also possible to customize your embedded Linux distribution with the help of graphical tools. TimeStorm provides graphical development tools for kernel configuration, driver development, file system configuration, and testing. Eclipse in combination with some plug-ins sets a new level of standardization, flexibility, robustness, reliability and productivity in developing embedded system [vH04b, vH04c].

3.5 TinyTD

TinyTD is like TimeStorm IDE an Eclipse-based IDE for developing TinyOS applications. It was supported by a IBM Eclipse Innovation Award because of some of the following features [SBD05]:

- Highlighting of nesC code [GLvB+03]
- Code navigation
- Code completion
- A component browser

- A configuration view
- Support for multiple TinyOS source trees
- Automatic build support
- Fully integrated TinyOS compiler toolchain

3.6 Generic Modeling Environment (GME) and GRATISII

People at the Vanderbilt University developed a graphical environment for model-based development of TinyOS applications under the name GRATIS II (Graphical Development Environment for TinyOS) [LD03]. It is a fully functional modeling, code generation, verification and parsing environment, which is based on GME [LMB+01].

GME allows to create and configure domain-specific modeling and program synthesis environments and generates inputs to Commercial Off-The-Shelf (COTS) analysis tools. The technology behind GME are metamodels, that specifies the modeling paradigm of the application domain. Because of domains without a large potential market, GME serves as a design environment that can be configured for a wide range of domains. Therefore, you need a generic concept which provides sufficient abstraction for this task. The advantage is obvious in only programming once such an environment and than configure it to your needs by ourself. Metamodels in GME that are specifying the modeling paradigm automatically generates the target-specific environment, which itself is used to build the metamodels. In this environment you can build domain models that are stored in a model database. Out of this database the application or input for different COTS are generated. To provide a powerful generic tool, GME can build large-scale, complex models. Models may includes Atoms, References, Connections, Constraints, and Sets and can be instantiated like classes in an object-oriented language. So it is easy to reuse and maintenance models in the whole hierarchy and build individual libraries for your domain. The technology behind all this is the Unified Modeling Language (UML) for the syntactic definitions of the metamodeling paradigm. Constraints are used to specify static semantics using the Object Constraint Language (OCL). For further details take a look at [LMB+01].

During another project at the Vanderbilt University called NEST (Network Embedded Systems Technology) several middleware services, tools and applications were developed as add-ons for GME. Add-ons can cooperate with GME using some or all events that are generated of GME. So you can extend the capabilities of GME, without modifying itself, writing your own add-on. One

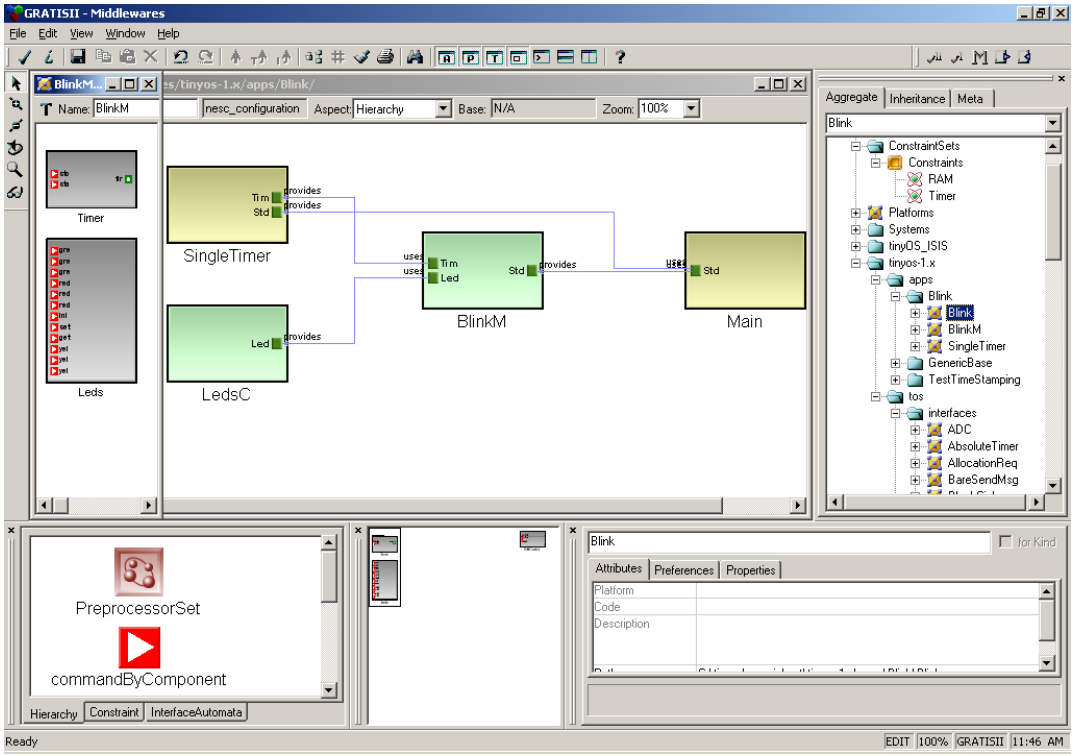

Figure 3.1: GRATIS II environment

add-on created by NEST was GRATIS II. GRATIS II is using TinyOS and nesC for generating code for embedded systems. GRATIS II can automatically parse the entire TinyOS tree and the corresponding graphical models to generate the code for the OS and the application. They developed this add-on for GME because of their research-field of large-scale distributed processing nodes with limited resources and tightly coupled sensors and actuators. Because of those strict constraints you have to use a thin application-specific operating system and middleware layers. GRATIS II is a model-based approach for developing applications based on TinyOS. It provides fully functional modeling, code generation and parsing based on GME [LD03].

We will give now a basic technical overview of GRATIS II. GME is used to create the metamodel for the graphical modeling language of GRATIS II. Furthermore OCL is used to set constraints for the models to keep them valid. The whole TinyOS v1.x source tree is already converted to graphical models through an effective parser add-on and can be viewed in the tree-browser. The parser can also optimize the generated library with cleanup, which means if there are inconsistencies in the files the parser will try to handle those errors as gracefully as possible. The second objective of the parser is auto placement, which means he tries to separate and organize all used, provided interfaces, con-

figurations and other nesC elements. Another add-on is the code generator for generating the complete application, which includes all necessary glue source-code files and also a Makefile, out of the models including the constraints of course. With GRATIS II you have an good graphical overview about all parts of TinyOS. You can also parse your own nesC (*.nc) files into a corresponding representation in GME [LD03].

3.7 Simulink

Simulink, provided by MathWorks, is a platform for model-based design combined with multidomain simulation. It includes an interactive graphical editor for assembling and managing intuitive block diagrams per drag and drop. Connections stand for mathematical relationships between the blocks. Simulink is based on MATLAB to have access to a huge number of tools for algorithm development, data visualization, data analysis, and access and numerical computation. Simulink provides hierarchical modeling by using subsystems, data management and model analysis and diagnostics tools to check the consistency of the model and ensure data integrity. Dynamic blocks like integration and unit delay, algorithmic blocks like sum and look up tables and structural blocks such as mux switches, and signal and bus selectors. More than 1000 blocks are available but you can also define your own blocks, including handwritten code, and import some add-ons, which makes Simulink a very powerful simulation tool. With hierarchical modeling you have the chance to choose the level of details of your model. Add more and more details if you want to implement your model. With control signals you can conditionally execute subsystems dynamically or even use the add-on *Stateflow* to model event-driven systems. Within the Model Explorer you can quickly set and change attributes of certain blocks or re purpose a model by selecting different data sets. For example, possible data types are single, double, singed, unsigned 8, 16 or 32 bit integers, fixedpoint, and Boolean. After your model is finished you can first and foremost simulate the dynamic behavior and view the results live. You can also set information to be mind during simulation. The results can be analyzed and visualized by viewing signals with the display and scopes of Simulink. There is a graphical debugger to diagnose unexpected behavior in the model. Simulink also provides tools for testing and validating your models. They help generating test conditions or check outputs of a block regarding to some constraints you set. The C code generator is not directly included with Simulink but comes in on add-on called Real-Time Workshop [Mat04d]. Real-Time Workshop generates ANSI/ISO C code and executables of all kind of models like developing and testing algorithms. The code is incrementally generated by using the model blocks and can be used for many real-time and non-real-time

applications. It is completely integrated into the Simulink environment. You can select and configure the target with the Model Explorer and save multiple configuration sets. Single-tasking and multitasking operating systems are supported as well as bare-board environments (without OS). You can create different kind of codes according to your field of application (efficient real-time execution, generate multiple instances of model code through dynamic memory allocation, etc.). For large-scale applications it is possible to generate code only for specific blocks or parts of the model to reduce the generation build times. Another part is code optimization. Real-Time Workshop provides the following kinds of code-efficiency [Mat04b]:

- Code reuse
- Expression folding
- Signal storage reuse
- Dead path elimination
- Parameter inlining
- Single-precision math libraries

The Real-Time Workshop Embedded Coder 4 generates production code for embedded systems. The most essential part in the field of embedded systems is to generate a exceptionally compact and fast code, also for on-target rapid prototyping boards, microprocessors and real-time operating systems (RTOS). To support targets with or without a RTOS in single, multitasking or asynchronous mode, the code must be target-independent. To improve the usability the code will be optimized and verified [Mat04c]. There are some add-ons for specific embedded targets to download the code directly to the microcontroller. After downloading it automatically begins execution in real time. It is possible to test and validate the code with Processor-in-the-Loop (PIL) testing. For example the MPC5xx [Mat04a] or TI C2000 [Mat06].

MATLAB and Simulink creates virtual environments based on system models, which are graphically developed using block diagrams. They are primarily used for dynamic system simulations and are becoming more and more the mainstream also in the field of automotive control systems. The V850 Integrated Development Environment works in conjunction with MATLAB to improve the development efficiency of control systems of automobiles and more. It uses the Real-Time Workshop Embedded Coder to generate auto-code (C source code automatically generated from a model in this case out of MATLAB and Simulink). The V850 IDE then automates the entire build process to download the code on the evaluation board. This all happens with a one-button-click. You can also customize the build process and debugger or simulation/emulation tools can be automatically started [NEC06].

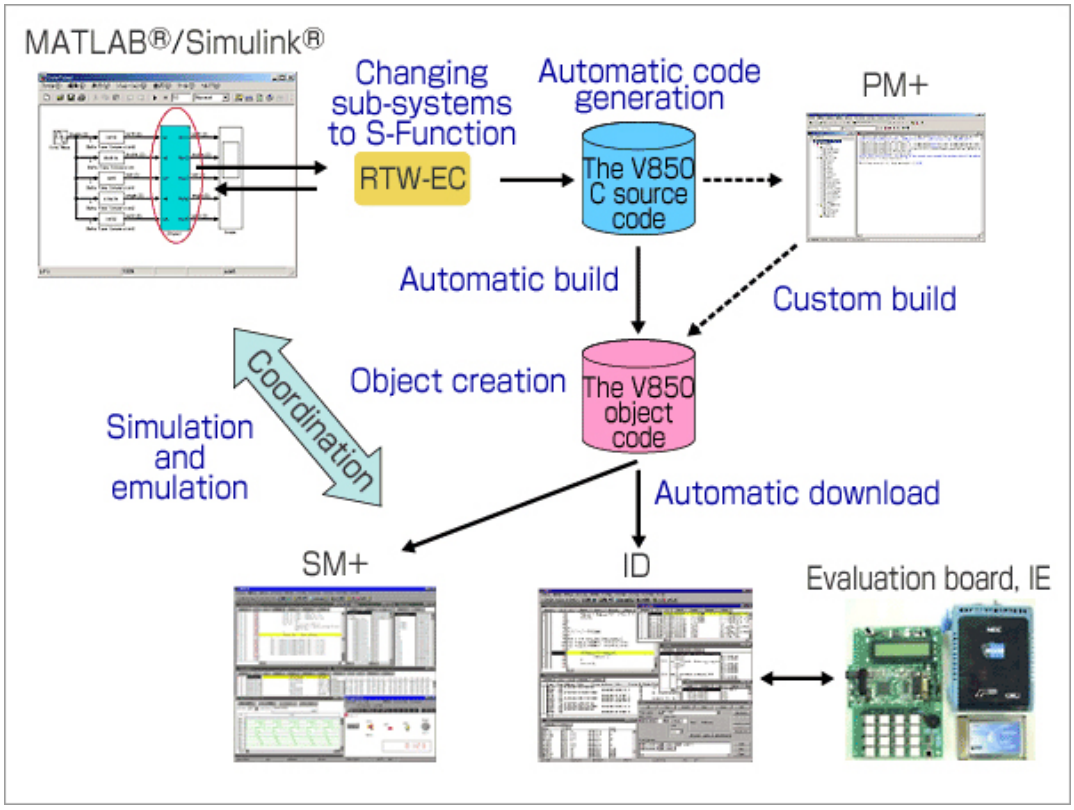

Figure 3.2: Simulink environment [NEC06]

3.8 LabVIEW

LabVIEW is a graphical development platform for design, control, and test provided by National Instruments. In 2003, LabVIEW expanded their product onto PDAs and FPGAs. 2005, they headed on towards the embedded area and DSPs. The newest attainment is a next step in cooperating projects with LEGO MINDSTORMS NXT.

- Complex tasks are visually broken down into insightful block diagrams
- A sequence of operations is logically captured in a flowchart-like manner
- Supported development platforms are Windows, Mac and Linux
- Supported target platforms are PCs, PDAs, real-time devices and embedded microprocessors

In LabVIEW you are programming code with a graphical user interface per drag and drop of different components called Virtual Instruments (VI) like measurement-objects, push-buttons, sliders, lamp indicators, graphs, etc. Configuration of those components works with property pages to set functionality and appearance. You can define the order of the VIs by wiring them together

and set the data flow. There are lots possibilities for debugging like showing the actual data flow, *comment out* code and Step-functions. Further you can create executables, shared libraries (DLLs) and installers.

LabVIEW can be used to acquire, analyze and present data. Examples for acquiring and analyzing are data acquisition, motion control, industrial communication, frequencies analysis, digital filtering, statistics, calculus, Fast Fourier Transform, differential equations and simulate signal. For presenting data there exists user interface objects, 3D object modeling, extensive charting and graphing utilities and also remote viewing and controlling of your application. You can create HTML and XML output and use third-party tools like MATLAB, Simulink, Maple, Word, Excel, etc. There are a lot of toolkits and modules available. For example signal processing, control design, simulation, instrument control, image acquisition, motion and vision, distributed monitoring, testing, etc.

You can also speed up the development process through reusable function libraries for hard real-time tasks [1].

LabVIEW provides several modules for special parts of development. The most interesting one within the scope of this thesis is the NI LabVIEW Embedded Development Module. It main features are [Nat05]:

- High-level graphical programming
- Lots of VIs for numerical analysis, signal processing, control, communications I/O driver and general-purpose logic
- Embedded Project Manager for target processors platforms
- debugging programs interactively from the front panel and block diagram
- Build in OCDI (on-chip debugging interface) for connection via JTAG, BDI, etc.
- Code generator for any 32-bit microprocessor toolchain/target

Developing with this module goes in two steps. The first is using the Project Manager for selecting a third-party toolchain and OS. The second step is to develop the embedded application graphically. You can automate the build process and build an executable by simply running the application. This executable can be simulate using WindRiver VxSim or download it directly onto the target. If your application is running on the target system you can send and receive data using a interactive front-panel on your PC or debug it using graphical block-diagrams. The advantage of this full-function graphical language is that also non experts at C-based programming can develop embedded systems application very easily [Nat05].

[1] http://www.ni.com/swf/labview/us/tour.htm

In cooperation with Analog Devices Inc. - National Instruments released in April 2006 a new module as extension for the graphical dataflow development environment - NI LabVIEW Embedded Module for ADI Blackfin Processors. It supports off-the-shelf measurement and control hardware for design, simulation, rapid prototyping, implementation, validation and verification of embedded systems all within one graphical development platform. It is a next step of an out-of-the-box, integrated solution for solving real-world problems within less time and without the need of professional, experienced embedded system programmers [Ins06].

Erik Goethert, design engineer at Bosten Engineering says: "Using NI LabVIEW Embedded technology, we have one tool to take the system model to hardware-in-the-loop for testing and prototyping all the way to the chip. This means we spend less time learning the details and syntax of traditional low-level tools and more time improving our designs."

Again math, analysis and signal processing functions are included as well as integrated I/O for video and audio DACs, ADCs and CODECs and of course real-time, interactive debugging [Ins06]. Dr. Fred Martin, assistant professor at the University of Massachusetts Lowell says about this productive learning environment: "LabVIEW Embedded technology makes robotics programming accessible to people who would not otherwise be able to create embedded systems. It gives users an alternative to programming in C. The LabVIEW graphical programming model is especially powerful for signal flow and signal processing applications and is much better than textual languages, especially for embedded design."

3.9 LEGO MINDSTORMS NXT

Although being a non-industrial, consumer electronic product, LEGO MINDSTORMS is the best example of simplifying embedded systems programming. The system is completely based on the LabVIEW graphical programming developing environment for automated measurement and control systems. The big difference is the simple, intuitive user interface of LEGO MINDSTORMS which allows also young people or novices programming embedded systems - in this case robots. The target group for LEGO MINDSTORMS is stated 10+. For simplicity reasons the system is not as powerful as LabVIEW but you can drag and drop blocks and features to react to sensors, give orders to actors and easily set the properties of each block. Depending on your MINDSTORM set there are touch sensors, sound sensor, light sensors, ultrasonic sensors, rotation sensors, lamps, gear motors available. With one click you can download and run the application on your NXT robot via USB or Bluetooth [MIN06].

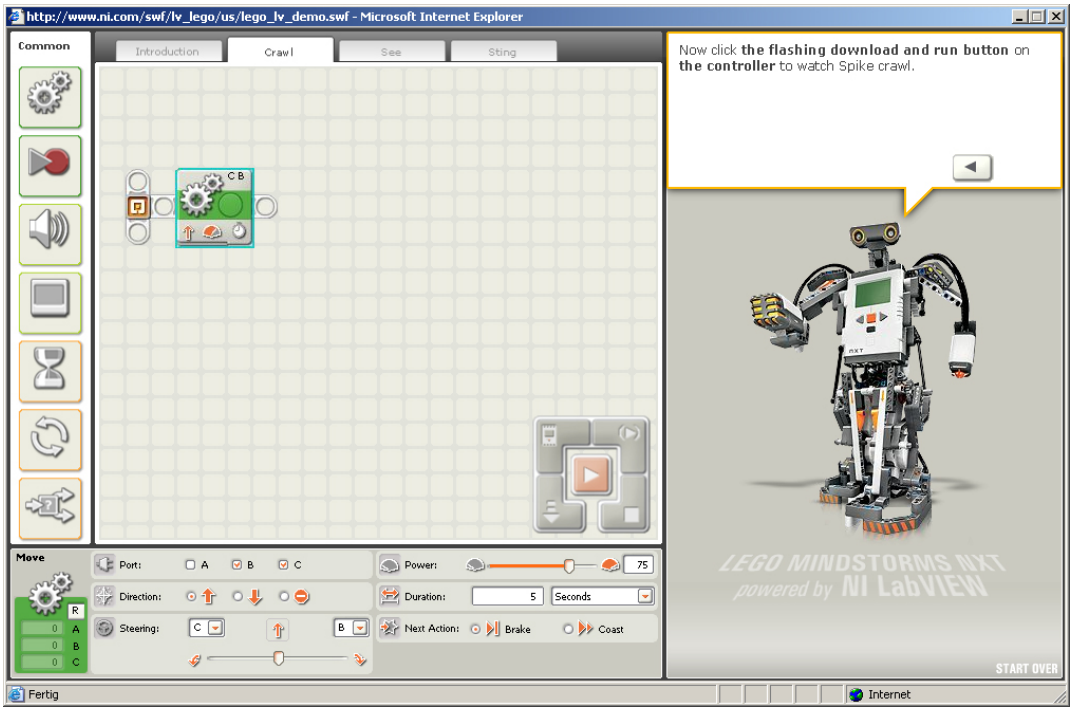

Figure 3.3: MINDSTORMS environment 1

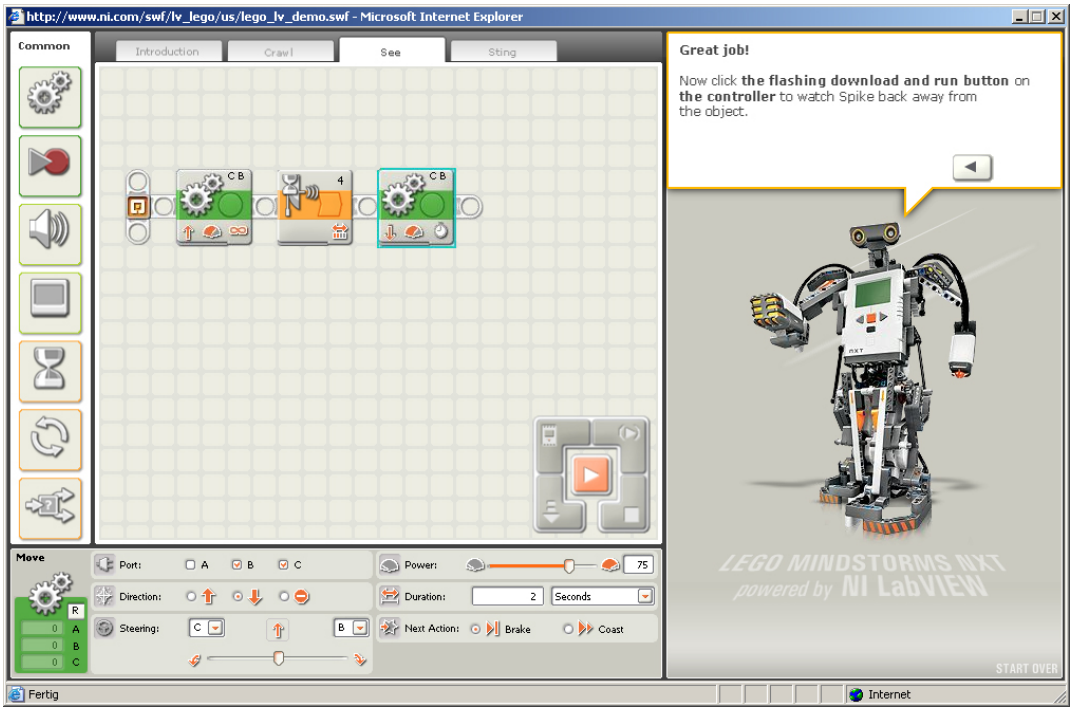

Figure 3.4: MINDSTORMS environment 2

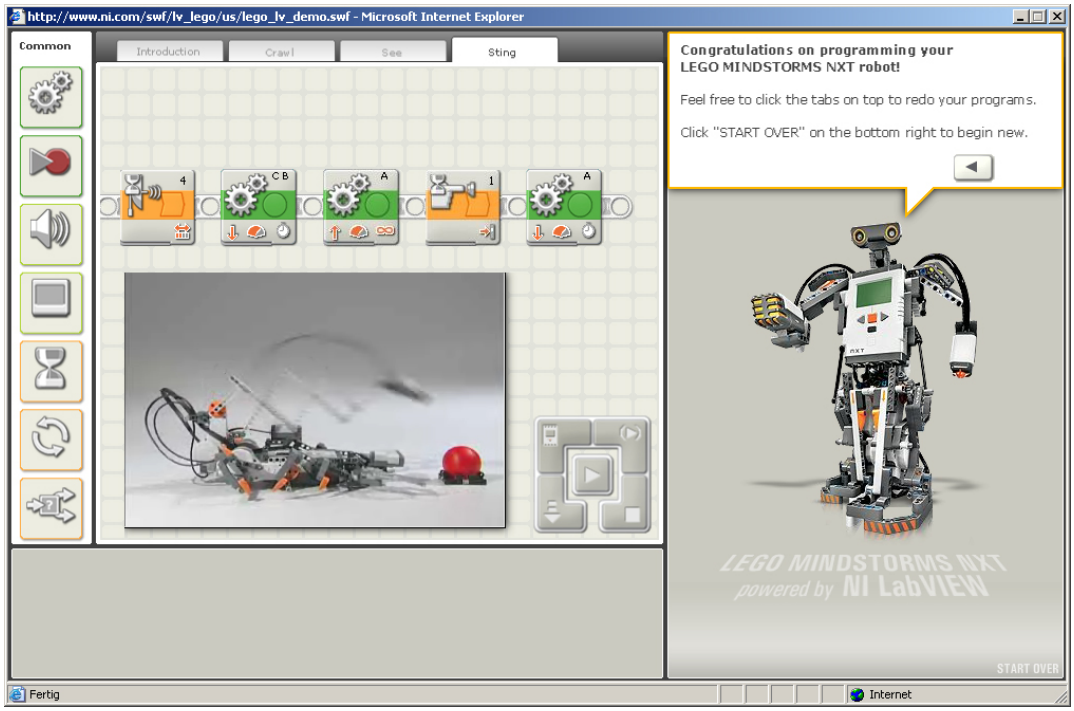

Figure 3.5: MINDSTORMS environment 3

3.10 Algorithmic builder

Algorithmic builder is a development environment providing simulation, debugging and chip programming of AVR microcontrollers. You enter the program as a flowchart with a tree-like structure and can use assembler level and macro level as well as multi-byte signed values. On the other hand there is no support for high level languages. It is easy to configure the chosen microcontroller and use the in-system programmer to download your program directly onto the chip. Before downloading you can simulate your program. It is also possible to use an On-Chip MONITOR to display the actual state of the microcontroller at predefined breakpoints. Again with the visualization the probability of errors will be reduced and, thus, development time will be shortened [http://algrom.net/].

3.11 Avidicy AVR C Compiler

Forest Electronic Developments (FED) offers AVIDICY as a front end development of AVR projects. In the application designer you can drag elements from a palette like timers, displays, ports, etc. and drop it on the processor. Further on you see the chosen processor with his pins, properties like oscillator frequency and can easily connect for example a port-element to the desired

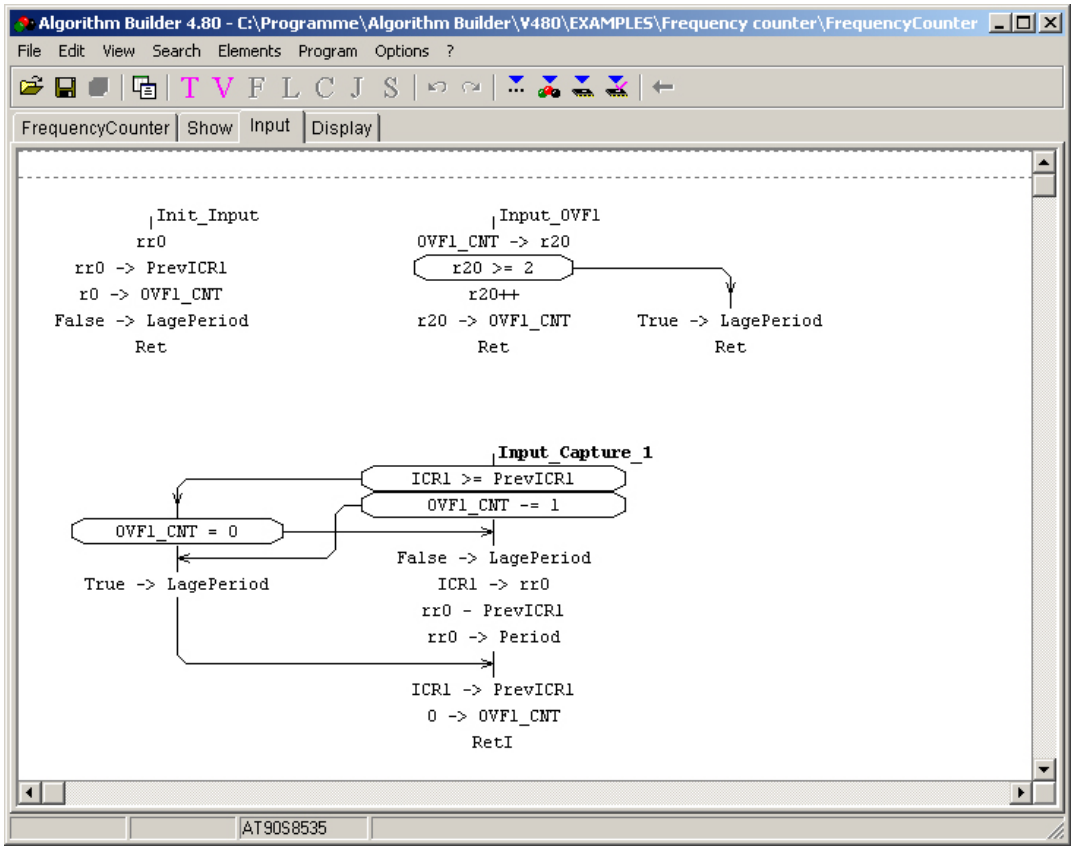

Figure 3.6: Algorithm Builder environment

pin. You can change the properties of each element by simple drop down lists, check-boxes etc. If a element uses events (interrupts), like a timer-overflow or the finished receiving of a byte over the serial interface, AVIDICY calls it Occurrences. In the property view you can enter the name of the function, which should be executed if the event occurs. The main application, initialization code and the main loop are automatically generated like initialization of timers, displays, ports, the bodies for functions of event-occurrences. But those functions have to be written by the user in C. So it is expected that users are familiar with programming in C.

Out of those C code files the machine code will be generated through the included FED AVR C Compiler. Another part of AVIDICY is AVRDESIM – a powerful simulation environment for AVR processors. For example view and change values of registers, ports, memory locations, set breakpoints, view number of processor machine cycles which have been executed, highest count reached by the watchdog timer, determine the execution of a certain block of code, use run, step and single step simulation and using stimulus and injection files to set a special state of your application only to mention a few features.

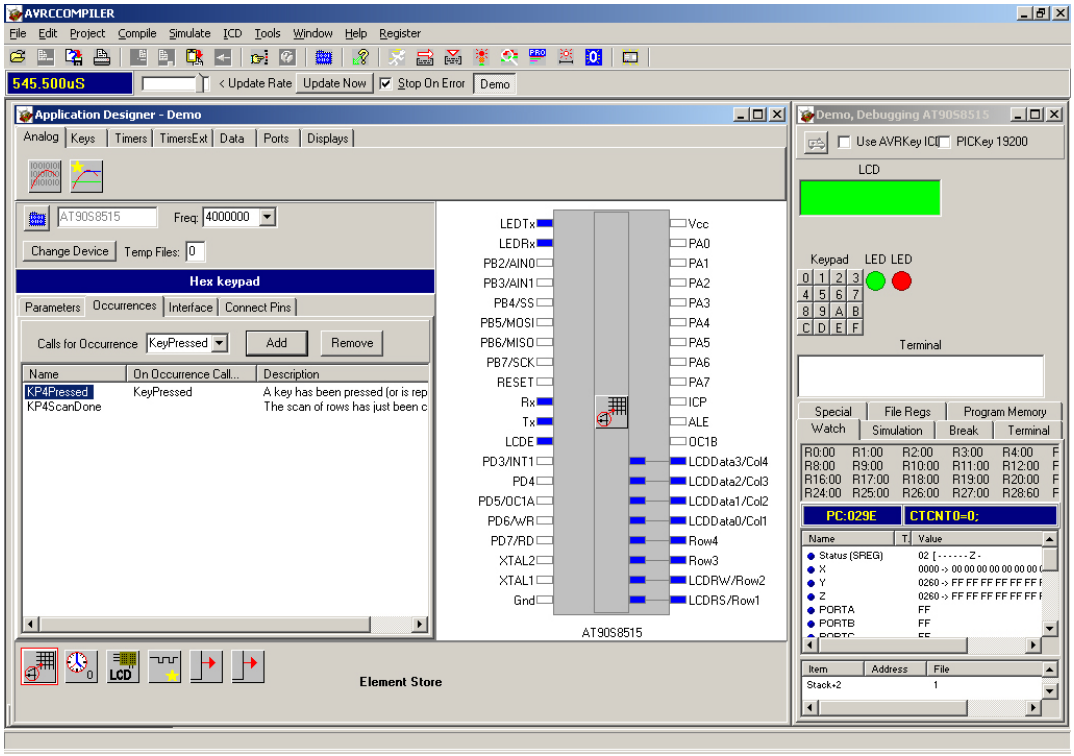

Figure 3.7: Avidicy environment 1

There is also a waveform analyzer (logic analyzer) of all pins of the processor available. After compiling you can start the debugger with external device simulation and test your application before downloading on the processor [Abb03].

The structure of the generated application is basically the following [Abb03, p. 26]:

```
void main()
{
        Initialization of registers, elements, ...
        Enable Interrupts
        Call function UserInitialise()          //USER CODE HERE

        while(1)
        {
                For Each Occurrence
                {
                        Test Occurrence Flag
                }

                Call functions associated with Occurrences
                Call any Element functions which are to be called regularly
                Call function UserLoop()        //USER CODE HERE

        }

}
```

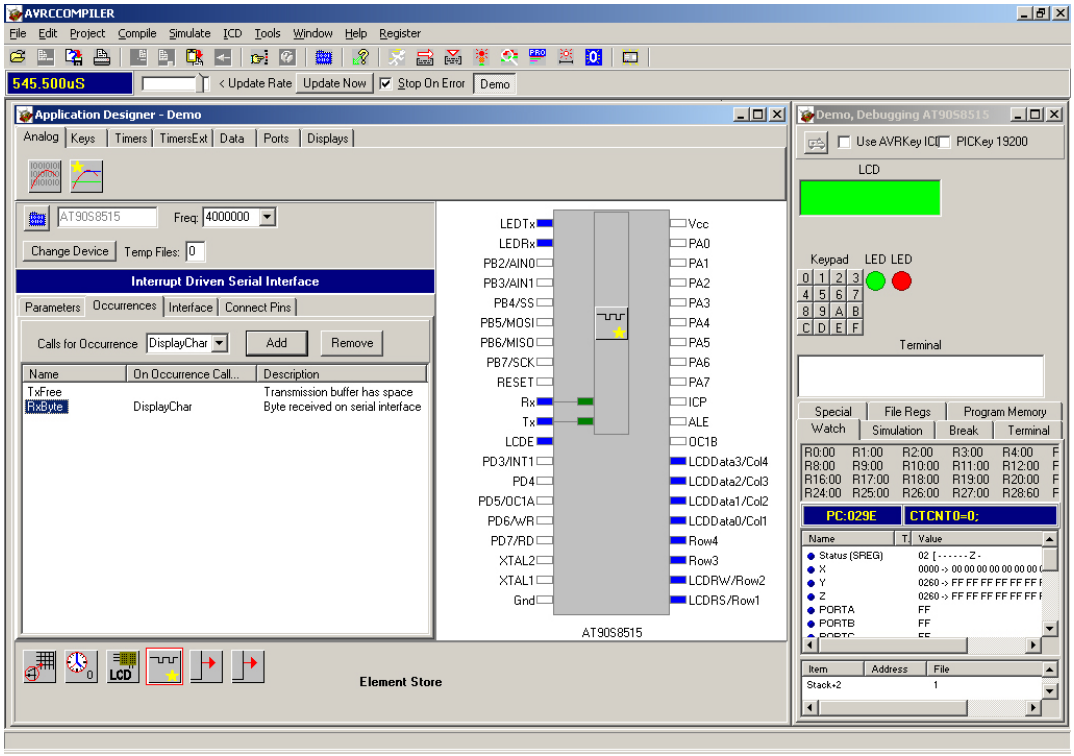

Figure 3.8: Avidicy environment 2

3.12 IAR Systems MakeApp and visualSTATE

IAR MakeApp helps you design and implement device drivers for microcontrollers through visual development tools to speed up this process. IAR MakeApp is for free and supports processors of Atmel, Philips and Renesas but it is only available for windows platforms. The used devices like microcontrollers, external devices and IP blocks are configured with a CAD-like drawing editor. The components are displayed graphically and the colour of the pins tells you if they are currently in use or not.

With a simple point-and-click mechanism you can generate device drivers for the supported microprocessors and external devices in ANSI C, which includes initialization, runtime control and interrupt handling functions. If you use IP blocks from the component library on the right side IAR MakeApp will generate source code as well for it. From this it follows that the values of the special function registers are calculate and set automatically according to the chosen properties. For each configurable component you can set those properties with the help of property dialogs. In the case of resource conflicts or illegal settings (illegal minimum, maximum values, duplicated use of multiplexed port pins or mutual exclusion features) the accurate rule checking mechanism flags those by

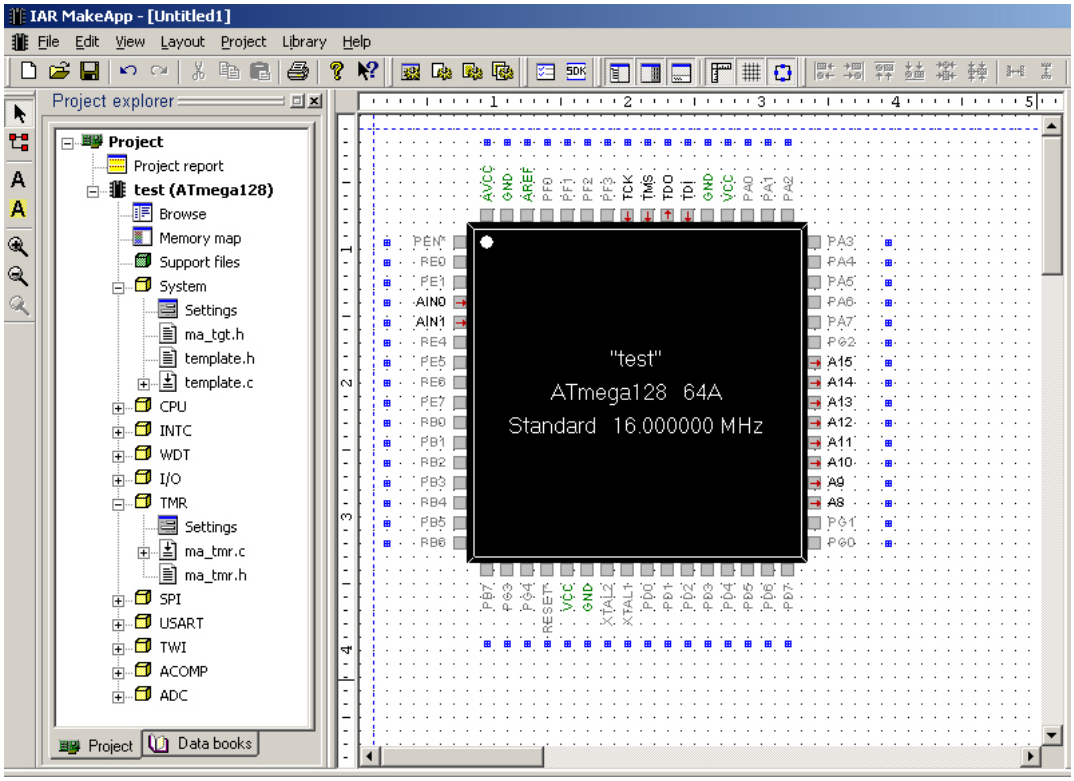

Figure 3.9: IAR MakeApp environment

alarm icons.

On the left side, in the project explorer, you see a tree view of all components, their modules, generated source files and functions, available in the current project. Supported peripherals:

- CPU - Bus control and memory
- INTC - Interrupt controller
- WDT - Watchdog timer
- I/O - I/O ports
- TMR - Timers, counters
- SPI - Serial peripheral interface
- USART - Universal synch/asynch serial communication
- TWI - Two-wire serial interface
- ACOMP - Analog comparator
- ADC - Analog to digital converter

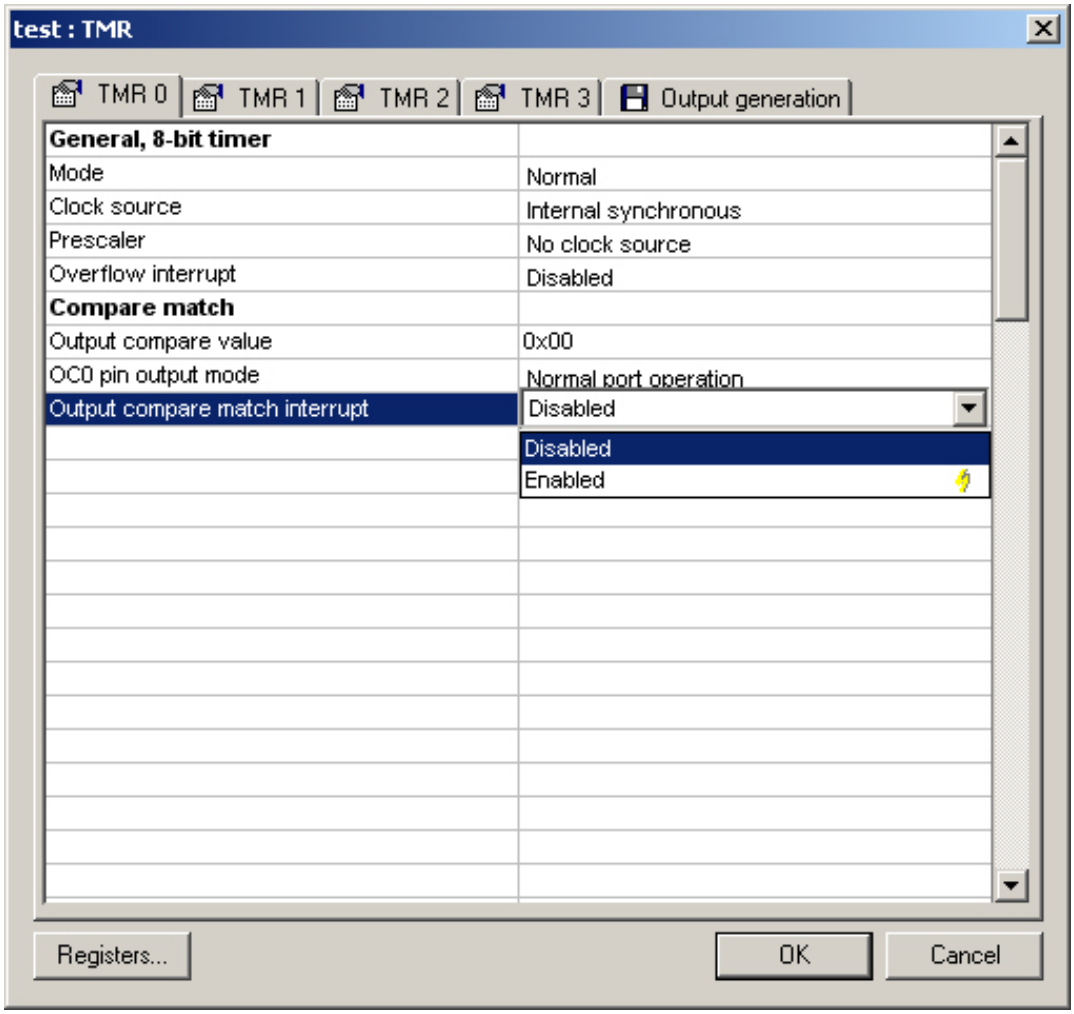

Figure 3.10: IAR MakeApp properties

The generated source code will be modified and optimized with the included code generator depending on your specified configurations to generate efficient and well-tested source code. Those output files will be ready to be used by our application software. Alongside the source code generation a extensive project report in HTML format will be created, which includes detailed information about chip resources (SFRs, pins, interrupts, ...), configurations, generated device driver functions, etc. But there are much more possibilities of automatically file generation. For example low-level SFR bit-field access macro files, high-level device driver functions, assembly source files, C source files, linker files, binary files, documentary, libraries and executables. You can also generate IAR visualSTATE integration files which will be mentioned in the next paragraph [www.iar.com/makeapp] [Sys01] [Tutorial user.pdf Seite 21ff IAR MakeApp].

visualSTATE is a graphical state machine design tool based on UML which is also suited for smaller development projects. It is based on a model-driven design and can be used during the whole development process. The visualization of the state machine makes it much easier to find logical mistakes and discuss with others about the design. It can contain more than one state machine which itself can be hierarchical. VisualSTATE verifies the design model and finds unreachable states, dead end states, *live lock* states, unused states, etc. Through a simulation you can visually validate your design and do regression testing and coverage testing before downloading on the target. But even on the target you can debug your design with RealLink. By pressing a single button you can convert your graphical design into a C or C++ implementation accordingly to your Atmel AVR processor and choose using a RTOS or not. The generated C/C++ code is very compact and reliable, comparable to handwritten code and can be tested very intensive. Integrating visualSTATE into ALTIA Faceplate allows you to create a GUI prototype very easily. And again an automatic creation of the documentation is supported [Sys05b, Sys05a].

The generated source files can be imported as a whole project into the IAR Embedded Workbench IDE to be edited. It also integrates the IAR C/C++ compiler, assembler, linker and C-SPY debugger. For more details take a look at [Sys06].

3.13 Comparison and Summary

To sum it up and show the differences between the applications presented in this section, we want to do a categorization and add a table to stress out some differences.

As we have seen in the examined applications it is important to have an environment where everything is build in, like TimeSys IDE and TinyTD. GRATIS II serves as good example on how to build a graphical model-based environment.

TinyTD, GRATIS II and LabVIEW are using some kind of operating system which makes it easy to develop an application executable on a wide range of microcontrollers. But it is better to be flexible deciding whether to use an OS or not like Real-Time Workshop is.

LabVIEW Embedded Module should be able to program any 32-bit microcontroller but currently does not support 8-bit processors. Like Simulink and MATLAB it is very powerful but in consequence rather complex and expensive.

Concerning usability the outstanding example is LEGO MINDSTORMS NXT because of its intuitive programming capabilities.

Simulink, LabVIEW, Algorithm builder, Avidicy and IAR Systems MakeApp provide more or less good testing, simulating and debugging capabilities. Therefore, the environment must be tightly coupled to the target system. For example Algorithm builder has a good balance of features which are nice to have and keeping a good overview on what is happening. For example, if changing the target CPU the application will map as many features as possible automatically and displays the remaining conflicts. The property settings are very intuitive and accordingly to your properties the Special Function Registers (SFRs) are set (Timers, Prescalers set in ms, CPU speed, initializations, ...). But the big disadvantages is that programming is only supported in a macro-level assembler. Changing this macro-level to a model-based block approach is necessary to support people in programming microcontrollers who are not experts in any program language.

3.13.1 Categorization

We can categorize the applications to simplify embedded system development into the following types:

1. Using a collection of different development tools manually (editor, compiler, linker, ...) or in an automated toolchain (e.g. *make*).
2. Using an integrated development environment (IDE) to control the whole development process with one framework. (e.g. TimeStorm IDE and TinyTD, or in general *.net* or *Eclipse*).
3. Using a graphical development environment (GDE) with all necessary tools in one framework:
   - But still have to write some code. (e.g. Algorithmic builder, Avidicy AVR C Compiler, IAR System MakeApp).
   - For analyzing and simulating. (e.g. MATLAB, Simulink).
   - For generating code for microcontrollers with some kind of running Operating System (OS) for embedded mostly 32-bit microcontrollers (e.g. GRATIS II which is based on GME and TinyOS, LabVIEW Embedded Development Module, LEGO MINDSTORMS NXT).
   - For generating code without using an OS for small microcontrollers. (e.g. the Real-Time Workshop add-on for Simulink).

3.13.2 Comparison Table

Name	Provider	Cost / Licenses	Output	Add hand written Code	Supported Development Platforms	Supported Target Platforms
GRAPE	grapesys, Israel	n/a anymore	machine code	yes	n/a	Motorola HC08, PIC-16Cxxx, PIC-16Fxxx, 8051 derivates
GRATIS II	Vanderbilt University	open source	nesC	n/a	Windows, Linux using Wine, MAC OS ?	processors supporting TinyOS
Simulink	MathWorks	15-day trial, student version: $ 99,–, commercial use: $ 2.800,–	only analyzing and simulating	yes	all	n/a
Real-Time Workshop Embedded Coder	MathWorks	no trial version, no student version, commercial use: $ 5.000,–	code for Embedded OS or machine code	yes	Windows, Mac OS, Linux	n/a
LabVIEW	National Instruments	different levels starting from $ 1.249,– up to $ 4.249,–	object oriented development	yes	Windows, Mac OS, Linux	n/a

Name	Provider	Cost / Licenses	Output	Add hand written Code	Supported Development Platforms	Supported Target Platforms
LabVIEW Embedded Module	National Instruments	$ 11.299,–	code for Embedded OS	yes	n/a	any 32-bit processor
LEGO MINDSTORMS NXT	LEGO, National Instruments	$ 249,–	n/a	no	Windows, Macintosh	32-bit ARM7 microprocessor
Algorithm builder	n/a	free	machine code	yes	Windows	Atmel processors
Avidicy AVR C Compiler	Forest Electronic Development (FED)	about $ 75,–	C code	yes	Windows	AVR family
IAR System MakeApp	IAR System	free	C code	yes	Windows	Atmel, Philips and Renesas processors

4 Design Approach

We decided to develop a new tool, called *Graphical Microcontroller Programing Environment(GMP)*. We want to support the following items, starting with a focus onto 8-bit AVR controllers:

- Simplifying the development process of embedded systems:
  - Avoid intrinsic complexity.
  - Non professionals should be able to program embedded systems.
  - No writing of any line of code.
  - Intuitive programming using graphical models.
  - The possibility to implement standard tasks as well as complex components.
  - Changing the target system easily.
- Speed up and focus onto the development process.
- Building an abstraction from the embedded target and target OS.
- Using a graphical environment.
- Using a dynamic model.

First of all we decided to use Eclipse as development platform like TimeStorm did because of it is cross platform compatibility and because it is opensource. So we got a stable, well tested framework for our development. Together with EMF and GEF it provides a powerful opensource platform including a modeling framework and a rich graphical editor. We did not choose GME, because it is written in C++ for Windows only. It runs in Linux using Wine[1], but we decided that this is not enough to build a new tool onto it. On the other hand, Eclipse is becoming the platform of choice in various areas of software development. Of course a lot of companies develop plugins for their processors only and accordingly to their special application needs (like Windriver, QNX, TimeSys, ...), but we want to go a step further. Our approach is to build a tool for developing programs for various kinds of embedded systems. Although we start with a specialization onto 8-bit microcontrollers of the AVR family, we want to be able to adopt to all kinds of microcontrollers. That makes it

[1] `http://de.wikipedia.org/wiki/WINE_Is_Not_an_Emulator`

necessary to have a dynamic model. The user has to be able to add his own controller descriptions on the fly to this model.

The next question is whether to choose an underlying OS or not. GRATIS II for example uses TinyOS or also LabVIEW Embedded Module generates C code for the chosen OS. With Real-Time Workshop you have the possibility to use or not to use an OS running on the processor. So the goal should be to use an OS if the user wants to. Since the 8-bit controllers have very limited resources and normally does not support any OS we decided to not use an OS for now.

The main goal of this thesis is to simplify the system level design of the embedded development process. We are developing an expert-in-a-box solution which can be extended by experts but provide an easy and intuitive user interface giving non-professionals the chance to easily create microcontroller programs. As it started with programming machine code, followed by Assembler and nowadays by C, we now come to a point which makes it possible for people to write programs for microcontrollers without writing a single line of code.

Integrated Development Environments (like TimeStorm IDE) create a connective link between writing, debugging, analyzing, downloading, and testing with different tools. An IDE integrate all necessary tools into one framework. For us it is important to provide such a framework which enables you to program your microcontroller with a model-based approach using a graphical interface mainly consisting of blocks and connections. Because microcontroller software is increasing in scale and complexity the development process must be more efficient, faster and cheaper. This contains the ability to easily change the target architecture, automatically find logical mistakes, resource conflicts (if a too small controller is chosen), and design errors.

Therefore we need a mechanism to map blocks and connections of the graphical program to the target hardware. This mapping should support the following items:

- Fast change of the target architecture
- Support for user specific block libraries
- Including the mapping of superblocks (= multiple blocks merged to a bigger one)
- Support for user specific hardware libraries including code-generators
- Automated search for inconsistencies and resource conflicts

The building process should be started using only one mouse click. It should generate the C code out of the graphical program, using a C compiler to compile

it into machine code, and download the generated binary to the target. The whole toolchain should be adoptable by the user.

Our design approach is based on libraries and code generators which only have to be written and developed once and can be reused. This avoids the need to recreate the same functionality again in each new design. You also do not need to know which bits and ports have to be set in a certain way and read the manuals again and again. To customize the blocks to the needs of the current application a properties view is used. With it you can set the behavior of each block. You only need professionals for developing that blocks and libraries for a microcontroller but afterwards users with a far lesser educational level can develop the applications. It should be possible to write the libraries for each available microcontroller.

Since implementing such a dynamic, powerful system needs a lot of work, we have decided to start with a small amount of blocks and a profound model-based design and dynamic environment. This tool can be expanded later with additional features like debugging, analyzing, downloading and testing capabilities.

5 Implementation

In this chapter we want to present the details of our implementation. We will start with all necessary tools you have to install to run the *Graphical Microcontroller Programing Environment(GMP)* followed by an short instruction on how to use it.

5.1 Details

We use Eclipse as platform because of the powerful plug-ins *EMF* and *GEF*. We use EMF to define our meta model including the meta information of our blocks and connections. We differ between data and event connections. Based on this model the different blocks are defined and grouped into libraries. Each library is saved within an XML file. This files must comply to a XML-schema and are loaded by the graphical environment and displayed within a palette. The editor is based on GEF since this simplifies the development of a graphical editor. Another point why we use EMF and GEF is, that they are developed in a manner that they can be easily used together. The GEF based editor serves as the viewer for our model. This model is also saved as XML file.

One of the most important properties of our implementation is flexibility. As already mentioned before this includes the block libraries. Furthermore it is possible to easily change the target hardware. This is possible because we use dynamic hardware libraries. Each library includes the definition of a microcontroller family with all its MCUs. The hardware library also includes the code generators.

Let us now have a deeper look to the block libraries. It is possible to define as many libraries as you want. Furthermore each library can contain as many blocks as desired. As an example we present the current version of our default library. It consits of a XML file containing the basic information on the blocks:

```
<?xml version="1.0" encoding="UTF-8"?>

<lib>
        <eventBlock name="OutPort">
                <inEventAnchor name="init" defaultInit="true"/>
                <inEventAnchor name="write">
                        <inDataAnchor name="data"/>
                </inEventAnchor>
```

```
                <outDataAnchor name="current"/>
        </eventBlock>
        <eventBlock name="Wait">
                <inEventAnchor name="init" defaultInit="true"/>
                <inEventAnchor name="wait">
                        <inDataAnchor name="duration"/>
                </inEventAnchor>
        </eventBlock>
        <dataPathBlock name="Ror">
                <inDataAnchor name="input"/>
                <outDataAnchor name="result"/>
        </dataPathBlock>
        <eventBlock name="Interval">
                <inEventAnchor name="init" defaultInit="true">
                        <inDataAnchor name="interval"/>
                </inEventAnchor>
                <outEventAnchor name="ready"/>
        </eventBlock>
</lib>
```

Another optional file can be added to the library directory containing display information like icons and block captions:

```
title=Default Library

OutPort.title=Output Port
OutPort.icon=
OutPort.init.title=Init
OutPort.write.title=Write
OutPort.write.data.title=Data
OutPort.current.title=Current Value

Wait.title=Busy Wait
Wait.init.title=Init
Wait.icon=
Wait.wait.title=Start
Wait.wait.duration.title=Duration

Ror.title=Rotate Right
Ror.icon=
Ror.input.title=Input
Ror.result.title=Result

Interval.title=Interval
Interval.icon=
Interval.init.title=Init
Interval.init.interval.title=Duration
Interval.ready.title=Event
```

We have now seen how the block libraries are defined. They are needed to create the basic models. The next step is the mapping to the actual target hardware. The mapping is based on the hardware libraries. We have added a simple example for one MCU of the AVR family. First we specify the properties common to all MCUs of the family in a general XML file:

```
<mcu id="avr_general">
        <component lib="at.ac.tuwien.vmars.gmp.default_lib" id="OPORT">
                <require type="port" subtype="direction:o" count="*"/>
        </component>

        <component lib="at.ac.tuwien.vmars.gmp.default_lib" id="Interval">
```

```
                <require type="port" count="1"/>
        </component>
</mcu>
```

The controller specific information is contained in another XML file:

```
<mcu id="atmega128" parent="avr_general">
        <component type="timer" id="TIMER0" subtype="width:8"/>
        <component type="timer" id="TIMER1" subtype="width:16"/>

        <package="qfp64">
                <component type="port" id="PORTA" count="8" subtype="direction
                  :io"/>
                <component type="port" id="PORTB" count="8" subtype="direction
                  :io"/>
                <component type="port" id="PORTC" count="8" subtype="direction
                  :io"/>
                <component type="port" id="PORTD" count="8" subtype="direction
                  :io"/>
                <component type="port" id="PORTE" count="8" subtype="direction
                  :io"/>
        </package>
</mcu>
```

As you can see, each block is associated with a generator in the hardware library. The code for some example generates can be found in appendix A.1.

5.2 Installation

To run our application you have to install eclipse together with the plug-ins EMF and GEF which are all available at the eclipse-homepage:

- `www.eclipse.org`
- `www.eclipse.org/emf`
- `www.eclipse.org/gef`

The next step is to import our code into your workbench and launch it as Eclipse Application. In the newly opened Eclipse instance create a new project and add two new simple files, one model file with the ending *.gmp* and one mapping file with the same name and the ending *.mapping*. When double clicking the model file it should normally be opened by the GMP Editor. If not, right click the file and choose *Open With-GMP Editor*. From the palette on the right side of the editor you can drag and drop blocks into it and draw connections between the blocks. By right click on a component you can duplicate its input anchors if necessary. To display the properties view choose *Window - Show View - Other* in the menu and select *Properties* from the category *General*. In the newly displayed view you can view and change the properties of the selected block (name, value, bus width, initialization order, ...).

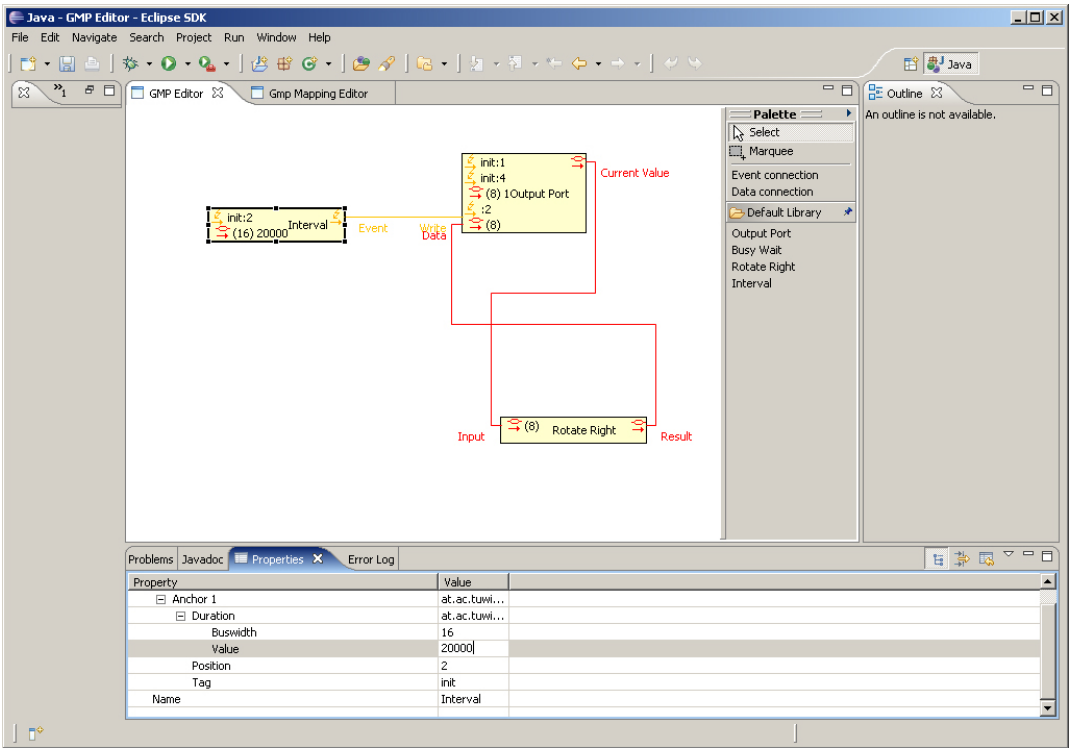

Figure 5.1: GMP editor with components palette

By double clicking the mapping file the mapping editor is opened. In the mapping file you can specify which hardware resources you want to use for your application. The resources needed by your model are automatically displayed in a table. After mapping all resources you can press the "'Generate Code"' button to create your application code. Currently this code is printed to the console of the eclipse application that started GMP.

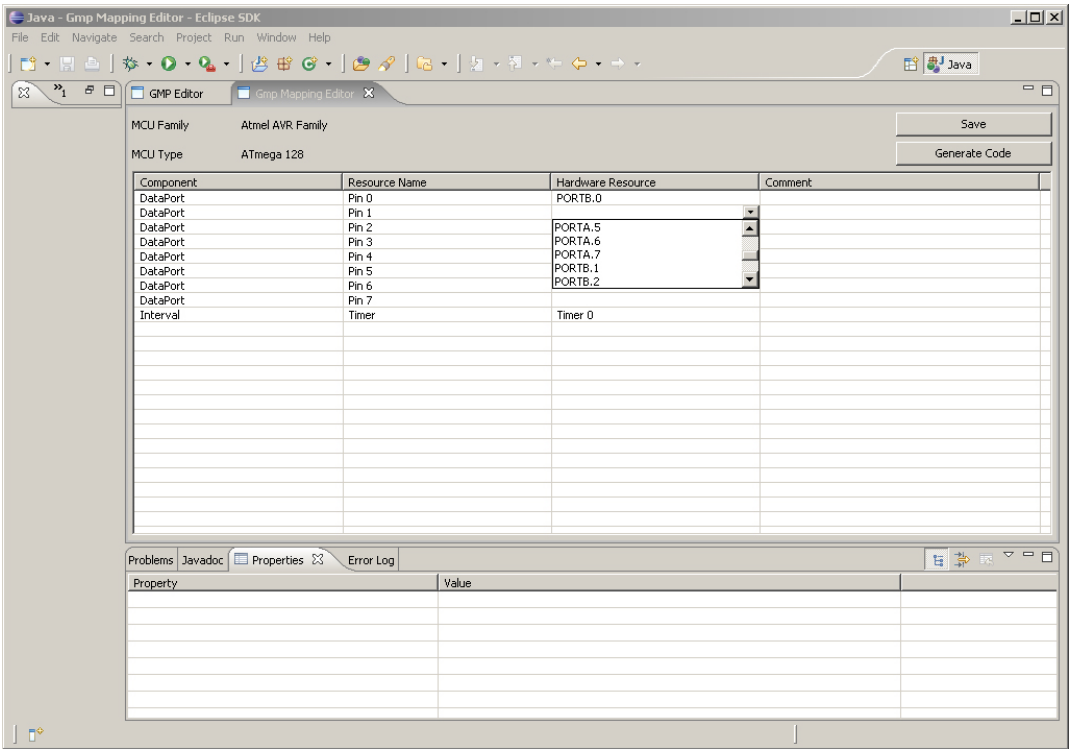

Figure 5.2: Mapping file of GMP

6 Results and Discussion

The most important part of the *Graphical Microcontroller Programing Environment (GMP)* is the dynamic model based approach. It should allow advanced users to build their own libraries for a certain processors, and should enable novice users to start programming their microcontrollers without writing a single line of code. Another important point is, that the created application is independent of the target platform. The target hardware is selected in the last step of the development phase. This enables you to change the MCU from a 8-bit architecture to an 32-bit one, by only changing the mapping. Furthermore the mapping functionality helps you mapping your application by automatically choose the components suitable to your needs and by detecting conflicts in the mapping like Algorithm builder does.

TimeSys IDE and TinyTD are examples of an environment where everything is build into one tool. Since they are based on Eclipse we decided to use it as well. We took GRATIS II as example for building a graphical model-based environment, but we choose EMF and GEF as basis for our implementation because of their platform independence.

We started with an implementation which does not use an underlying operating system. This is a big difference to the other tools like TinyTD, GRATIS II and LabVIEW. In the future we want to be as flexible as Real-Time Workshop when it comes to using an OS or not.

LabVIEW Embedded Module and Simulink are very powerful but in consequence rather complex and expensive. Therefore the usability of our tool should be as intuitive for the user as LEGO MINDSTORMS NXT.

Simulink, LabVIEW, Algorithm builder, Avidicy and IAR Systems MakeApp provide more or less good testing, simulating and debugging capabilities. During this thesis it was not possible to implement any of this features. But since their importants in embedded design it is a big goal for the future to add this features in a sufficient extend.

As a long time goal it should be realistic to implement the possibility to create state machines and even distributed embedded systems based for example on TTP/A.

7 Conclusion

With the *Graphical Microcontroller Programing Environment(GMP)* we want to help novices to get into to field of programing embedded systems without writing any line of code. Another important goal is to help advanced users to simplify, speed up and focus on their development process. Therefore *GMP* abstracts from the embedded target and OS and uses a graphical environment based on a dynamic model. This model can be used to build own libraries and develop standard tasks but also more complex components. The mapping functionality of *GMP* makes changing the target CPU no big issue.

The current implementation proves, that our approach is implementable and useable. The current version can be seen as prototype. The next steps will be the implementation of more components, the implementation of more target independence, adding a wider range of target systems, building in graphical debugging capabilities and add the functionality to develop distributed systems. Those steps will be turned into practice in following projects and a master thesis.

A Appendix

A.1 Codegenerator Wait

```
package at.ac.tuwien.vmars.gmp.atmega_lib.generators;

import java.util.LinkedList;
import java.util.List;

import at.ac.tuwien.vmars.gmp.model.mapping.interfaces.Entry;
import at.ac.tuwien.vmars.gmp.model.mapping.interfaces.Mapping;
import at.ac.tuwien.vmars.gmp.model.partlibrary.EventBlockGenerator;
import at.ac.tuwien.vmars.gmp.model.partlibrary.interfaces.McuType;
import at.ac.tuwien.vmars.gmp.model.runtime.interfaces.BaseBlock;
import at.ac.tuwien.vmars.gmp.model.runtime.interfaces.EventBlock;
import at.ac.tuwien.vmars.gmp.model.runtime.interfaces.InEventAnchor;
import at.ac.tuwien.vmars.gmp.model.runtime.interfaces.OutDataAnchor;

public class WaitGenerator extends EventBlockGenerator
{
  public WaitGenerator(McuType mcu)
  {
    super(mcu);
  }

  public void generate(EventBlock block, Mapping mapping, List<String> output,
     List<String> prototypes)
  {
    prototypes.add("void␣start" + block.getName() + "(" + getDatatypeForWidth
      (((InEventAnchor)block.getInEventAnchor("wait").getAnchors().get(0)).
      getInDataAnchor("duration").getWidth()) + "␣duration);");
    output.add("void␣start" + block.getName() + "(" + getDatatypeForWidth(((
      InEventAnchor)block.getInEventAnchor("wait").getAnchors().get(0)).
      getInDataAnchor("duration").getWidth()) + "␣duration)");
    output.add("{");
    output.add("\tuint8_t␣i;");
    output.add("\tfor(i␣=␣0;␣i␣<␣duration;␣i++)");
    output.add("\t\t_delay_ms(1);");
    output.add("}");
  }

  public String getEventAnchorConnectData(InEventAnchor anchor)
  {
    if(anchor.getDefinition().getName().equals("init"))
      ; //Not needed on this plattform
    else if(anchor.getDefinition().getName().equals("wait"))
      return "start" + anchor.getContainer().getBlock().getName();
    else
      ;//TODO: Error Handling
        return null;
  }
```

```
  private String getDatatypeForWidth(int width)
  {
    if (width <= 0)
      throw new RuntimeException(); // TODO: Errorhandling
    else if (width <= 8)
      return "uint8_t";
    else if (width <= 16)
      return "uint16_t";
    else if (width <= 32)
      return "uint32_t";
    else
      throw new RuntimeException(); // TODO: Errorhandling
  }

  public String getDataAnchorConnectData(OutDataAnchor anchor)
  {
    return null;
  }

  public List<Entry> deserializeEntries(BaseBlock block)
  {
    List<Entry> entries = new LinkedList<Entry>();
    return entries;
  }
}
```

A.2 Codegenerator Ror

```
package at.ac.tuwien.vmars.gmp.atmega_lib.generators;

import java.util.LinkedList;
import java.util.List;

import at.ac.tuwien.vmars.gmp.model.mapping.interfaces.Entry;
import at.ac.tuwien.vmars.gmp.model.mapping.interfaces.Mapping;
import at.ac.tuwien.vmars.gmp.model.partlibrary.BlockGenerator;
import at.ac.tuwien.vmars.gmp.model.partlibrary.DatapathBlockGenerator;
import at.ac.tuwien.vmars.gmp.model.partlibrary.interfaces.McuType;
import at.ac.tuwien.vmars.gmp.model.runtime.interfaces.BaseBlock;
import at.ac.tuwien.vmars.gmp.model.runtime.interfaces.DatapathBlock;
import at.ac.tuwien.vmars.gmp.model.runtime.interfaces.OutDataAnchor;


public class RorGenerator extends DatapathBlockGenerator
{
  public RorGenerator(McuType mcu)
  {
    super(mcu);
  }

  public void generate(DatapathBlock block, Mapping mapping, List<String>
    output, List<String> prototypes)
  {
    int width = block.getInDataAnchor("input").getWidth();
    prototypes.add(getDatatypeForWidth(width) + "␣calculate" + block.getName()
       + "(" + getDatatypeForWidth(width) +"␣value);");
    output.add(getDatatypeForWidth(width) + "␣calculate" + block.getName() + "
      (" + getDatatypeForWidth(width) +"␣value)");
    output.add("{");
    output.add("\tvalue␣=␣value␣>>␣1;");
    output.add("\tif(value␣==␣0)");
```

```
    output.add("\t\tvalue␣=␣" + (1 << (width - 1)) + ";");
    output.add("\treturn␣value;");
    output.add("}");
  }

  public String getDataAnchorConnectData(OutDataAnchor anchor)
  {
    if(anchor.getDefinition().getName().equals("result"))
    {
      String result = "calculate" + anchor.getBlock().getName() + "(";

      OutDataAnchor source = ((DatapathBlock)anchor.getBlock()).
        getInDataAnchor("input").getConnection().getSource();

      try
      {
        BlockGenerator generator = getMcu().getGenerator().getGenerator(source
          .getBlock().getDefinition());
        result += generator.getDataAnchorConnectData(source);
      }
      catch(Exception ex)
      {
        ex.printStackTrace();
        // TODO: Handle Exception
      }

      result += ")";
      return result;
    }
    else
      return null;
  }

  private String getDatatypeForWidth(int width)
  {
    if (width <= 0)
      throw new RuntimeException(); // TODO: Errorhandling
    else if (width <= 8)
      return "uint8_t";
    else if (width <= 16)
      return "uint16_t";
    else if (width <= 32)
      return "uint32_t";
    else
      throw new RuntimeException();
  }

  public List<Entry> deserializeEntries(BaseBlock block)
  {
    List<Entry> entries = new LinkedList<Entry>();
    return entries;
  }
}
```

A.3 Codegenerator Interval

```
package at.ac.tuwien.vmars.gmp.atmega_lib.generators;

import java.util.LinkedList;
import java.util.List;
```

```
import at.ac.tuwien.vmars.gmp.model.mapping.interfaces.Entry;
import at.ac.tuwien.vmars.gmp.model.mapping.interfaces.Mapping;
import at.ac.tuwien.vmars.gmp.model.mapping.metadata.MappingFactory;
import at.ac.tuwien.vmars.gmp.model.partlibrary.BlockGenerator;
import at.ac.tuwien.vmars.gmp.model.partlibrary.EventBlockGenerator;
import at.ac.tuwien.vmars.gmp.model.partlibrary.interfaces.McuType;
import at.ac.tuwien.vmars.gmp.model.runtime.interfaces.BaseBlock;
import at.ac.tuwien.vmars.gmp.model.runtime.interfaces.EventBlock;
import at.ac.tuwien.vmars.gmp.model.runtime.interfaces.EventConnection;
import at.ac.tuwien.vmars.gmp.model.runtime.interfaces.InDataAnchor;
import at.ac.tuwien.vmars.gmp.model.runtime.interfaces.InEventAnchor;
import at.ac.tuwien.vmars.gmp.model.runtime.interfaces.OutDataAnchor;
import at.ac.tuwien.vmars.gmp.model.runtime.interfaces.OutEventAnchor;

public class IntervalGenerator extends EventBlockGenerator
{
  public IntervalGenerator(McuType mcu)
  {
    super(mcu);
  }

  public void generate(EventBlock block, Mapping mapping, List<String> output,
     List<String> prototypes)
  {
    try
    {
    int width = ((InEventAnchor)block.getInEventAnchor("init").getAnchors().
      get(0)).getInDataAnchor("interval").getWidth();
    prototypes.add("void␣init" + block.getName() + "(" + getDatatypeForWidth(
      width) + "␣interval);");
    output.add("void␣init" + block.getName() + "(" + getDatatypeForWidth(width
      ) + "␣interval)");
    output.add("{");
    output.add("\tTCCR1B␣=␣(1␣<<␣WGM12)␣|␣(1␣<<␣CS12)␣|␣(1␣<<␣CS10);");
    output.add("\tOCR1A␣=␣interval;");
    output.add("\tTIMSK␣|=␣(1␣<<␣OCIE1A);");
    output.add("}");
    output.add("SIGNAL(SIG_OUTPUT_COMPARE1A)");
    output.add("{");
    OutEventAnchor anchor = block.getOutEventAnchor("ready");
    List<EventConnection> connections = anchor.getConnection();
    int maxPosition = anchor.getMaxPosition();
    for (int i = 0; i <= maxPosition; i++)
    {
      for (EventConnection connection : connections)
      {
        InEventAnchor targetAnchor = connection.getTarget();
        if (targetAnchor.getPosition() == i)
        {
          EventBlockGenerator targetGenerator = getMcu().getGenerator().
            getEventBlockGenerator(targetAnchor.getContainer().getBlock().
            getDefinition());
          String out = "\t" + targetGenerator.getEventAnchorConnectData(
            targetAnchor) + "(";
          boolean first = true;
          for (InDataAnchor dataAnchor : (List<InDataAnchor>)targetAnchor.
            getInDataAnchors())
          {
            if (dataAnchor.getConnection() == null)
              out += ((first) ? "" : ",␣") + "0x" + Integer.toHexString(
                dataAnchor.getConstantValue()).toUpperCase();
            else
            {
```

```
            OutDataAnchor outDataAnchor = dataAnchor.getConnection().
              getSource();
            BlockGenerator generator2 = getMcu().getGenerator().getGenerator
              (outDataAnchor.getBlock().getDefinition());
            out += generator2.getDataAnchorConnectData(outDataAnchor);
          }
          first = false;
        }
        out += ");";
        output.add(out);
      }
    }
  }
  output.add("}");
  }
  catch(Exception ex)
  {
    ex.printStackTrace();
    //TODO: Handle Exception
  }
}

public String getDataAnchorConnectData(OutDataAnchor anchor)
{
  return null;
}

public String getEventAnchorConnectData(InEventAnchor anchor)
{
  if (anchor.getDefinition().getName().equals("init"))
    return "init" + anchor.getContainer().getBlock().getName();

  return null;
}

private String getDatatypeForWidth(int width)
{
  if (width <= 0)
    throw new RuntimeException(); // TODO: Errorhandling
  else if (width <= 8)
    return "uint8_t";
  else if (width <= 16)
    return "uint16_t";
  else if (width <= 32)
    return "uint32_t";
  else
    throw new RuntimeException(); // TODO: Errorhandling
}

public List<Entry> deserializeEntries(BaseBlock block)
{
  List<Entry> entries = new LinkedList<Entry>();
  Entry entry = MappingFactory.eINSTANCE.createEntry();
  entry.setLibraryBlock(block);
  entry.setTitle("Timer");
  entry.setMappingKey("timer");
  entry.setBlockTypeKey("timer");
  entries.add(entry);
  return entries;
}
}
```

A.4 Codegenerator Port

```
package at.ac.tuwien.vmars.gmp.atmega_lib.generators;

import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;

import at.ac.tuwien.vmars.gmp.model.mapping.interfaces.Entry;
import at.ac.tuwien.vmars.gmp.model.mapping.interfaces.Mapping;
import at.ac.tuwien.vmars.gmp.model.mapping.metadata.MappingFactory;
import at.ac.tuwien.vmars.gmp.model.partlibrary.EventBlockGenerator;
import at.ac.tuwien.vmars.gmp.model.partlibrary.interfaces.McuType;
import at.ac.tuwien.vmars.gmp.model.runtime.interfaces.BaseBlock;
import at.ac.tuwien.vmars.gmp.model.runtime.interfaces.EventBlock;
import at.ac.tuwien.vmars.gmp.model.runtime.interfaces.InDataAnchor;
import at.ac.tuwien.vmars.gmp.model.runtime.interfaces.InEventAnchor;
import at.ac.tuwien.vmars.gmp.model.runtime.interfaces.OutDataAnchor;

public class PortGenerator extends EventBlockGenerator
{
  public PortGenerator(McuType mcu)
  {
    super(mcu);
  }

        public void generate(EventBlock block, Mapping mapping, List<String>
          output, List<String> prototypes)
        {
                generateInit(block, mapping, output, prototypes);
                generateEnable(block, mapping, output, prototypes);
                generateCurrent(block, mapping, output, prototypes);
        }

        private void generateCurrent(EventBlock block, Mapping mapping, List<
          String> output, List<String> prototypes)
        {
                InDataAnchor dataAnchor = ((InEventAnchor) block.
                  getInEventAnchor("write").getAnchors().get(0)).
                  getInDataAnchor("data");
                int width = dataAnchor.getWidth();

                prototypes.add(getDatatypeForWidth(width) + "␣get" + block.
                  getName() + "Current();");
                output.add(getDatatypeForWidth(width) + "␣get" + block.getName
                  () + "Current()");
                output.add("{");
                output.add("\tuint8_t␣portValue;");
                output.add("\t" + getDatatypeForWidth(width) + "␣value␣=␣0;");
                for (String portName : getUsedPortNames(mapping))
                {
                        buildCurrentValue(width, portName, getUsedPortIndices(
                          mapping, portName), output);
                        output.add("\treturn␣value;");
                }
                output.add("}");
        }

        private void generateInit(EventBlock block, Mapping mapping, List<
```

```
  String> output, List<String> prototypes)
{
        InDataAnchor dataAnchor = ((InEventAnchor) block.
          getInEventAnchor("write").getAnchors().get(0)).
          getInDataAnchor("data");
        int width = dataAnchor.getWidth();
        prototypes.add("void␣init" + block.getName() + "();");
        output.add("void␣init" + block.getName() + "()");
        output.add("{");
        for (String portName : getUsedPortNames(mapping))
        {
                int ddr = 0;
                Map<Integer, Integer> mappings = getUsedPortIndices(
                  mapping, portName);
                for (Integer portIndex : mappings.keySet())
                {
                        if (mappings.get(portIndex) < width)
                                ddr += (int) Math.pow(2, portIndex);
                }
                output.add("\tDDR" + portName + "␣|=␣0x" + Integer.
                  toHexString(ddr).toUpperCase() + ";");
        }
        output.add("}");
}

private void generateEnable(EventBlock block, Mapping mapping, List<
  String> output, List<String> prototypes)
{
        InDataAnchor dataAnchor = ((InEventAnchor) block.
          getInEventAnchor("write").getAnchors().get(0)).
          getInDataAnchor("data");
        int width = dataAnchor.getWidth();
        prototypes.add("void␣output" + block.getName() + "(" +
          getDatatypeForWidth(width) + "␣data);");
        output.add("void␣output" + block.getName() + "(" +
          getDatatypeForWidth(width) + "␣data)");
        output.add("{");
        output.add("\tuint8_t␣portValue;");
        for (String portName : getUsedPortNames(mapping))
        {
                int portMask1 = 0xFF;
                int portMask2 = 0;
                Map<Integer, Integer> mappings = getUsedPortIndices(
                  mapping, portName);
                for (Integer portIndex : mappings.keySet())
                {
                        if (mappings.get(portIndex) < width)
                        {
                                portMask1 -= (int) Math.pow(2,
                                  portIndex);
                                portMask2 += (int) Math.pow(2,
                                  portIndex);
                        }
                }
                output.add("\tportValue␣=␣PORT" + portName + "␣&␣0x" +
                   Integer.toHexString(portMask1).toUpperCase() + ";")
                  ;
                buildPortValue(width, getUsedPortIndices(mapping,
                  portName), output);

                output.add("\tPORT" + portName + "␣=␣portValue;");
        }
        output.add("}");
```

```
    }

    private String getDatatypeForWidth(int width)
    {
            if (width <= 0)
                    throw new RuntimeException(); // TODO: Errorhandling
            else if (width <= 8)
                    return "uint8_t";
            else if (width <= 16)
                    return "uint16_t";
            else if (width <= 32)
                    return "uint32_t";
            else
                    throw new RuntimeException(); // TODO: Errorhandling
    }

    private Set<String> getUsedPortNames(Mapping mapping)
    {
            Set<String> ports = new HashSet<String>();

            for (Entry entry : (List<Entry>)mapping.getEntries())
            {
  if(entry.getBlockTypeKey().equals("port"))
  {
    int dotIndex = entry.getHardwareBlock().getKey().indexOf(".");
    ports.add(entry.getHardwareBlock().getKey().substring(0, dotIndex));
  }
            }

            return ports;
    }

    private Map<Integer, Integer> getUsedPortIndices(Mapping mapping,
      String portName)
    {
            Map<Integer, Integer> ports = new HashMap<Integer, Integer>();

for (Entry entry : (List<Entry>)mapping.getEntries())
{
  if(entry.getBlockTypeKey().equals("port"))
  {
    int dotIndex = entry.getHardwareBlock().getKey().indexOf(".");
                            if (entry.getHardwareBlock().getKey().
                              substring(0, dotIndex).equals(portName))
                            {
      int dotIndex2 = entry.getMappingKey().indexOf(".");
                                    ports.put(Integer.parseInt(entry.
                                      getHardwareBlock().getKey().
                                      substring(dotIndex + 1)), Integer.
                                      parseInt(entry.getMappingKey().
                                      substring(dotIndex2 + 1)));
                                    // TODO: Errorhandling for non
                                      integers
                            }
                    }
            }

            return ports;
    }

    private void buildPortValue(int width, Map<Integer, Integer> mappings,
       List<String> output)
    {
```

```
        for (Integer portIndex : mappings.keySet())
        {
                if (mappings.get(portIndex) < width)
                {
                        int shift = mappings.get(portIndex) -
                          portIndex;
                        if (shift < 0)
                                output.add("\tportValue␣|=␣(data␣&␣(1␣
                                  <<␣" + mappings.get(portIndex) + "))
                                  ␣<<␣" + (-1 * shift) + ";");
                        else if (shift > 0)
                                output.add("\tportValue␣|=␣(data␣&␣(1␣
                                  <<␣" + mappings.get(portIndex) + "))
                                  ␣>>␣" + shift + ";");
                        else
                                output.add("\tportValue␣|=␣data␣&␣(1␣
                                  <<␣" + mappings.get(portIndex) + ");
                                  ");
                }
        }
}

private void buildCurrentValue(int width, String portName, Map<Integer
  , Integer> mappings, List<String> output)
{
        output.add("\tportValue␣=␣PORT" + portName + ";");

        for (Integer portIndex : mappings.keySet())
        {
                if (mappings.get(portIndex) < width)
                {
                        int shift = mappings.get(portIndex) -
                          portIndex;
                        if (shift < 0)
                                output.add("\tvalue␣|=␣(portValue␣&␣(1
                                  ␣<<␣" + portIndex + "))␣>>␣" + (-1 *
                                   shift) + ";");
                        else if (shift > 0)
                                output.add("\tvalue␣|=␣(portValue␣&␣(1
                                  ␣<<␣" + portIndex + "))␣<<␣" + shift
                                   + ";");
                        else
                                output.add("\tvalue␣|=␣portValue␣&␣(1␣
                                  <<␣" + portIndex + ");");
                }
        }
}

public String getEventAnchorConnectData(InEventAnchor anchor)
{
        if (anchor.getDefinition().getName().equals("init"))
                return "init" + anchor.getContainer().getBlock().
                  getName();
        else if (anchor.getDefinition().getName().equals("write"))
                return "output" + anchor.getContainer().getBlock().
                  getName();
        else
                ; // TODO: Errorhandling

        return null;
}

public String getDataAnchorConnectData(OutDataAnchor data)
```

```
        {
                return "get" + data.getBlock().getName() + "Current()";
        }

  public List<Entry> deserializeEntries(BaseBlock block)
  {
    List<Entry> entries = new LinkedList<Entry>();

    if(block instanceof EventBlock)
    {
      InDataAnchor dataAnchor = ((InEventAnchor) ((EventBlock)block).
        getInEventAnchor("write").getAnchors().get(0)).getInDataAnchor("data")
        ;
      int width = dataAnchor.getWidth();

      for(int i=0; i < width; i++)
      {
        Entry entry = MappingFactory.eINSTANCE.createEntry();
        entry.setLibraryBlock(block);
        entry.setTitle("Pin␣" + i);
        entry.setMappingKey("" + i);
        entry.setBlockTypeKey("port");
        entries.add(entry);
      }
    }

    return entries;
  }
}
```

Bibliography

[Abb03] Robin Abbott. *AVIDICY Rapid Application development for the AVR Microcontroller - Demonstration Manual*. Forest Electronic Developments, Hampshire, 2003. Comes with the demo download of AVIDICY AVR-C.

[GLvB+03] David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of Programming Language Design and Implementation*, June 2003. Available at `http://nescc.sourceforge.net/papers/nesc-pldi-2003.pdf`.

[Ins06] National Instruments. NI LabVIEW Embedded Design Platform Now Available for Analog Devices Blackfin Processors. Technical report, April 2006. Available at `http://digital.ni.com/worldwide/bwcontent.nsf/web/all/EABD67E44E43F4828625713F005CDED6`.

[LD03] Akos Ledeczi and Sebestyen Dora. GRATIS II. 2003. Available at `http://www.isis.vanderbilt.edu/projects/nest/gratis/index.html`.

[LMB+01] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing*, Nashville, USA, May 2001. Available at `http://www.isis.vanderbilt.edu/Projects/gme/GME2000Overview.pdf`.

[Ltd02] D. S. Grape Ltd. First commercial version of the grape embedded system development tool announced. April 2002. Available at `http://www.us.design-reuse.com/news/news3047.html`.

[Mat04a] Mathworks. Embedded Target for Motorola MPC555 - Deploy embedded code onto the Motorola MPC555. 2004. Available at `http://www.mathworks.com/mason/tag/proxy.html?dataid=4413&fileid=20677`.

[Mat04b] Mathworks. Real-Time Workshop 6.1 - Generate optimized, portable, and customizable code from Simulink mod-

els. 2004. Available at `http://www.mathworks.com/mason/tag/proxy.html?dataid=4429&fileid=20594`.

[Mat04c] Mathworks. Real-Time Workshop Embedded Coder 4 - Generate production code for embedded systems. 2004. Available at `http://www.mathworks.com/mason/tag/proxy.html?dataid=4412&fileid=21768`.

[Mat04d] Mathworks. Simulink 6 - Simulation and model-based design. 2004. Available at `http://www.mathworks.com/mason/tag/proxy.html?dataid=4429&fileid=20594`.

[Mat06] Mathworks. Embedded Target for the TI TMS320C2000 DSP Platform 2 - Deploy embedded code onto TI C2000 processors. 2006. Available at `http://www.mathworks.com/mason/tag/proxy.html?dataid=7146&fileid=31156`.

[MIN06] LEGO MINDSTORMS. How LEGO MINDSTORMS NXT Works. 2006. Available at `http://www.ni.com/academic/mindstorms/works.htm`.

[Nat05] National Instruments. Labview for embedded development. Technical report, 2005. Available at `http://www.ni.com/pdf/products/us/2005-5554-821-101-LO.pdf`.

[NEC06] NEC. The V850 Integrated Development Environment in Conjunction with MATLAB: Improving the Development Efficiency of Control Systems for Automobiles and More. In *NEC Electronics*, volume 53, February 2006. Available at `http://www.necel.com/en/channel/vol_0053/vol_0053_1.html`.

[SBD05] Janos Sallai, Gyorgy Balogh, and Sebestyen Dora. Tinydt. 2005. Available at `http://www.tinydt.net/`.

[Sys01] IAR Systems. *IAR MakeApp User Guide*. IAR Systems, August 2001. Comes with the demo download of IAR MakeApp.

[Sys05a] IAR Systems. IAR visualSTATE for Atmel AVR - visual programming tool dedicated for Atmel AVR. 2005. Available at `ftp://ftp.iar.se/WWWfiles/avr/ds-VSAVR.pdf`.

[Sys05b] IAR Systems. State machines for embedded systems. 2005. Available at `ftp://ftp.iar.se/WWWfiles/vs/DS-VS-2005-1.pdf`.

[Sys06] IAR Systems. Iar embedded workbench for the atmel avr microcontrollers. 2006. Available at `ftp://ftp.iar.se/WWWfiles/avr/DS-EWAVR-420.pdf`.

[vH04a] William von Hagen. Application development with eclipse-based ides. *LinuxDevices*, June 2004. Available at `http://linuxdevices.com/articles/AT4905486769.html`.

[vH04b] William von Hagen. Next-generation embedded linux development tools for high reliability and mission-critical applications. *Embedded Computing Design*, July 2004. Available at `http://www.embedded-computing.com/articles/id/?1935`.

[vH04c] William von Hagen. Simplifying embedded linux development with graphical tools. *LinuxDevices*, June 2004. Available at `http://linuxdevices.com/articles/AT4574262276.html`.